Inside WebObjects

JavaServer Pages and Servlets

November 2002

Contents

Figures, Listings, and Tables

CHAPTER 1

About This Book

JavaServer Pages (JSP) and servlets are important parts of Sun's J2EE (Java 2 Platform, Enterprise Edition) architecture. JSP is a specification that defines interfaces that servlet-container vendors can implement to provide developers the ability to create dynamic Web pages, which are files with the extension `.jsp`. Servlet containers interpret these files and create servlets (also know as workhorse servlets) to process HTTP requests and produce responses. Servlets are server plug-ins that extend the capabilities of your HTTP server. They provide a straightforward deployment mechanism for your applications. Servlets are deployed inside servlet containers, which are plug-ins to your HTTP server.

You should read this book if you want to deploy your WebObjects applications inside a servlet container or want to take advantage of WebObjects components (both standard and custom) in your JSP pages.

Deploying WebObjects applications as servlets allows you to take advantage of the features that your servlet container provides. Keep in mind that deployment tools such as Monitor and wotaskd do not work with servlets. WebObjects uses version 2.2 of the Servlet API, and version 1.1 of the JSP specification.

The book addresses two major points, each contained in its own chapter:

- Chapter 2, "Servlets" (page 11), explains how you develop WebObjects applications to be deployed as servlets and how to add servlet capability to existing applications.
- Chapter 3, "JavaServer Pages" (page 27), tells you how to write JSP-based applications, which can be thought of as JSP applications that use WebObjects technology or hybrids—applications that use JSP pages to accomplish some tasks and WebObjects components or direct actions to perform others.

- Appendix A, "Special Issues" (page 49), addresses special issues to consider when you deploy WebObjects applications as servlets or when you develop JSP-based applications.
- Appendix B, "Document Revision History" (page 53), lists the revisions made to this document.

To get the most out of this book, you must be familiar with WebObjects application development. In particular, you need to know how to create applications using Project Builder and how to layout WebObjects components using WebObjects Builder.

If you need to learn the basics about developing WebObjects applications, you can find that information in the following books:

- *Inside WebObjects: WebObjects Overview* provides you with a survey of WebObjects technologies and capabilities.
- *Inside WebObjects: Web Applications* shows you how to develop HTML-based applications.
- *Inside WebObjects: Deploying Applications* describes how to use WebObjects tools to deploy your applications as standalone entities.

For additional WebObjects documentation and links to other resources, visit http://developer.apple.com/webobjects.

In addition to WebObjects development experience, you also need to be acquainted with the syntax used in JSP pages and with the layout of WAR (Web Application Archive) files. You can find information about JSP and J2EE in the following books:

- *Java Servlet Programming*, 2nd edition (O'Reilly) provides an in-depth treatise on servlets. You can find more information at http://java.oreilly.com.
- *J2EE Technology in Practice* (Sun) provides an overview of J2EE technology.
- *JavaServer Pages Technology Syntax* (Sun) is a short document that describes the syntax used in JSP pages. You can download it from http://java.sun.com/products/jsp/technical.html. For more information on JSP and servlets, see http://java.sun.com/products/jsp.
- *Java Servlet Technology* contains the latest information on Sun's Java Servlet technology. You can view it at http://java.sun.com/products/servlet/.

WebObjects Developer also includes a commented, application project that shows you how JSP pages can take advantage of WebObjects components and direct actions. The example—using the client/server approach—includes two WebObjects application projects named SchoolToolsClient and SchoolToolsServer.You can find the projects in the `/Developer/Examples/JavaWebObjects` directory.

The three servlet containers supported in WebObjects are listed in Table 1-1.

Table 1-1 Servlet containers supported in WebObjects

Platform	Container	Version
Mac OS X Server	Tomcat	3.2.4
Solaris	WebLogic	7.0
Windows 2000	WebSphere	4.0.4

About This Book

C H A P T E R 2

Servlets

Servlet technology was developed as an improvement over CGI. It's an open standard that can be freely adopted by any vendor. It provides an infrastructure that allows applications from different manufactures to cooperate, and share resources.

The following sections explain how you can take advantage of servlet technology in WebObjects:

- "Servlets in WebObjects" (page 12) provides an overview of servlet technology as it is implemented in WebObjects.
- "Developing a Servlet" (page 13) guides you through creating a simple servlet.
- "Deploying a Servlet" (page 15) explores deployment issues and task you need to keep in mind when deploying a servlet.
- "Adding Servlet Support to an Existing Application" (page 19) explains how to add servlet support to an existing WebObjects application.
- "Servlet Single Directory Deployment" (page 21) describes the feature that allows you to create a directory containing the files necessary to deploy an application as a servlet that does not require WebObjects to be installed on the deployment computer.
- "Cross-Platform Deployment" (page 22) shows you how to simplify cross-platform deployment (or deployment in a platform other than the development platform) by allowing you to easily define the paths your servlet container uses to locate WebObjects frameworks, local frameworks, and WebObjects application bundles—WebObjects Application (WOA) directories.

Servlets in WebObjects

Servlets are generic server extensions that expand the functionality of a server. By deploying WebObjects applications as servlets running inside servlet containers, you can take advantage of the features that your servlet container offers. Alternatively, you can deploy your applications using an HTTP adaptor that runs as a plug-in in your HTTP server. The adaptor forwards requests to your servlet container.

WebObjects applications can be deployed as servlets inside a servlet container such as Tomcat, WebLogic, or WebSphere. When an application runs as a servlet, instead of as a separate Java virtual machine (JVM) process, it runs inside the servlet container's JVM, along with other applications. Note, however, that you can run only one instance of an application inside a servlet container. To run multiple instances of an application, you have to use multiple servlet containers. In addition, WebObjects deployment tools such as Monitor and wotaskd cannot be used with servlets.

To deploy an application as a servlet, you need to add the JavaWOJSPServlet framework to your project. When you build the project, Project Builder generates a WAR (Web application archive) file in addition to the WOA (WebObjects application) bundle. The WAR file has the appropriate classes and the `web.xml` file in the `WEB-INF` directory that your servlet container needs to launch the servlet. All you need to do in order to deploy the servlet is copy the WAR file to the application deployment directory of your servlet container.

You may have to modify `web.xml.template`, specifically the `%WOClassPath%` marker, to ensure that the classpath to the application's WOA is correct. For WebLogic, the default Session class must be placed in a package because it conflicts with an internal WebLogic class. In general, all your classes should be inside packages.

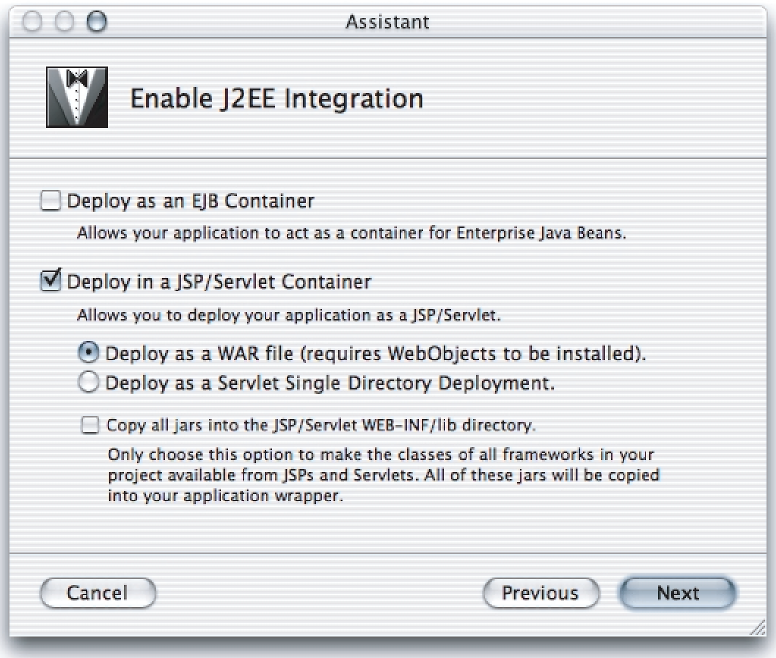
The WAR file is not a complete application. WebObjects Deployment must be installed on the application host, as well as the application's WOA bundle. However, using the Servlet Single Directory Deployment feature, you can deploy directories that contain all the necessary WebObjects classes. For more information, see "Servlet Single Directory Deployment" (page 21).

Developing a Servlet

This section shows you how to create a simple servlet using Project Builder.

Start by creating a WebObjects application project named `Hello`. You can deploy other types of WebObjects applications as servlets, such as Direct to Java Client, Direct to Web, Display Group, and Java Client.

In the Enable J2EE Integration pane of the Assistant, select Deploy in a JSP/Servlet Container.



Enable J2EE Integration

☐ Deploy as an EJB Container

Allows your application to act as a container for Enterprise Java Beans.

☑ Deploy in a JSP/Servlet Container

Allows you to deploy your application as a JSP/Servlet.

● Deploy as a WAR file (requires WebObjects to be installed).

○ Deploy as a Servlet Single Directory Deployment.

☐ Copy all jars into the JSP/Servlet WEB-INF/lib directory.

Only choose this option to make the classes of all frameworks in your project available from JSPs and Servlets. All of these jars will be copied into your application wrapper.

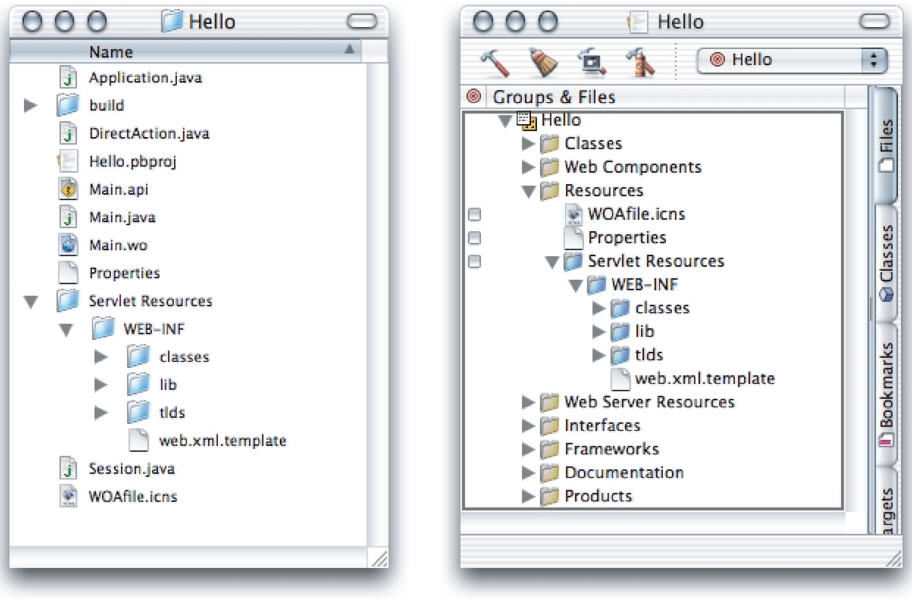
The “Deploy as a WAR file” option tells Project Builder to create WAR file, which should be placed in your servlet container’s application directory. The WAR file contains all the files needed by an application except WebObjects frameworks. Therefore, WebObjects needs to be installed on the computer on which you want to deploy the application.

The Deploy as a Servlet Single Directory Deployment option tells Project Builder to include WebObjects frameworks in the WAR file. With this option WebObjects does not need to be installed on the deployment computer.

The “Copy all JARs inside the JSP/Servlet WEB-INF directory” option tells Project Builder to copy framework and application JAR files to the `WEB-INF/lib` directory (necessary only when the servlet uses other servlets, or for JSPs that make use of actual objects).

As the right side of Figure 2-1 shows, the newly created project is, in all respects, a standard WebObjects application project. However, Project Builder adds the Servlet Resources folder to the Resources group. Anything you add to this folder is included in the WAR file or single deployment directory that Project Builder creates when you build the project, following the same directory structure. The Servlet Resources folder is a real directory in the project’s root directory; it’s shown on the left side of Figure 2-1.

Figure 2-1 Hello project directory and Project Builder window



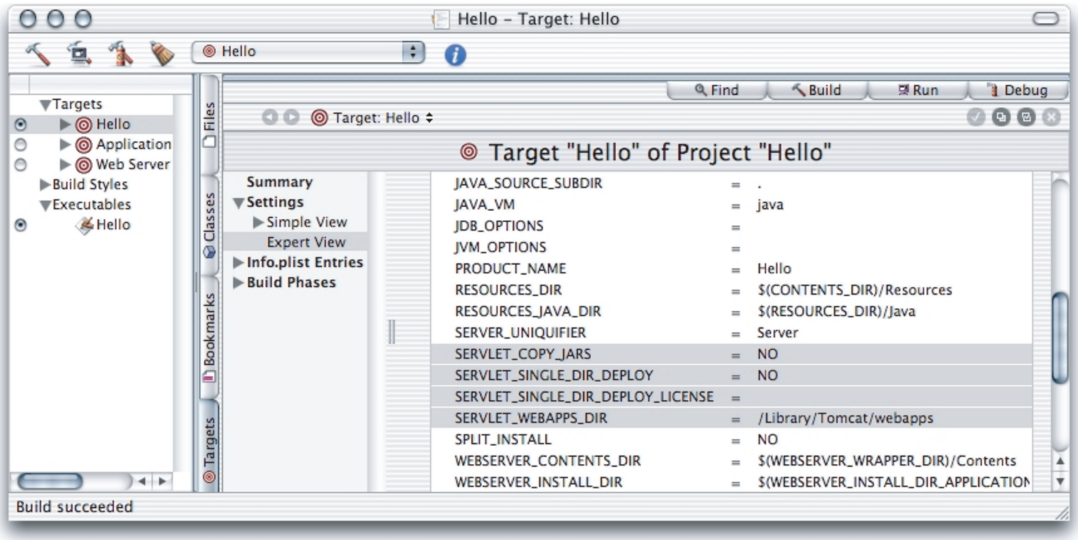
Deploying a Servlet

The WEB-INF folder, under Server Resources, contains the `web.xml.template` file, which Project Builder uses to generate the servlet's deployment descriptor. You can edit this template to customize the deployment descriptor for your deployment environment. There are several elements whose values are surrounded by percent (%) characters (these are placeholders that Project Builder evaluates when you build the project). These elements include cross-platform settings (see "Cross-Platform Deployment" (page 22) for details). You can replace the placeholders with other values if your environment requires it.

Follow these steps to get to the JSP and servlet build settings in Project Builder:

1. Click Targets, then click the Hello target in the Targets list. The Target pane appears. It contains the target settings list and a content pane.
2. Click Expert View under Settings in the target settings list to display the Hello target's build settings in the content pane.
3. Locate the `SERVLET_WEBAPPS_DIR` build setting and enter the path of your servlet container's application directory, as shown in Figure 2-2.

Figure 2-2 Build settings for a servlet project



JAVA_SOURCE_SUBDIR	=	.
JAVA_VM	=	java
JDB_OPTIONS	=	
JVM_OPTIONS	=	
PRODUCT_NAME	=	Hello
RESOURCES_DIR	=	$(CONTENTS_DIR)/Resources
RESOURCES_JAVA_DIR	=	$(RESOURCES_DIR)/Java
SERVER_UNIQUIFIER	=	Server
SERVLET_COPY_JARS	=	NO
SERVLET_SINGLE_DIR_DEPLOY	=	NO
SERVLET_SINGLE_DIR_DEPLOY_LICENSE	=	
SERVLET_WEBAPPS_DIR	=	/Library/Tomcat/webapps
SPLIT_INSTALL	=	NO
WEBSERVER_CONTENTS_DIR	=	$(WEBSERVER_WRAPPER_DIR)/Contents
WEBSERVER_INSTALL_DIR	=	$(WEBSERVER_INSTALL_DIR_APPLICATION

The `SERVLET_COPY_JARS` build setting tells Project Builder whether to copy framework and application JAR files to the `WEB-INF/lib` directory (necessary only when the servlet uses other servlets, or for JSPs that make use of actual objects).

The `SERVLET_SINGLE_DIR_DEPLOY` build setting indicates whether the application is to be deployed as a WAR file or a single deployment directory (see "Servlet Single Directory Deployment" (page 21) for more information). Set it to `NO` to deploy as a WAR file and `YES` to deploy as a single deployment directory.

The SERVLET_SINGLE_DIR_DEPLOY_LICENSE build setting must contain your WebObjects Deployment license when SERVLET_SINGLE_DIR_DEPLOY is set to YES. If you don't add your deployment license, you will not be able to build the application.

You can tell Project Builder where to put the WAR file by setting the value of the SERVLET_WEBAPPS_DIR build setting (this is especially convenient during development). By default, WAR files are places in the build directory of your project.

Project Builder WO (on Windows) adds two buckets to your project: JSP Servlet WEB-INF and JSP Servlet Resources. The JSP Servlet WEB-INF bucket is a holding place for JAR files, classes, and TLD files (which are auto-routed to the correct subdirectories in the WEB-INF directory of the generated WAR file or single deployment directory, lib, class, and tld respectively; the web.xml.template file is also located here). The JSP Servlet Resources bucket contains any other items you want to add to the WAR file or single deployment directory (you can drag files and folders into this bucket; Project Builder WO preserves the directory structure when it generates the WAR file). These items are not auto-routed.

There are also several new variables defined in Makefile.preamble. The SERVLET_APP_MODE variable indicates whether Web-server resources are loaded from the WOA bundle (the default) or the servlet container (by setting it to "Deployment". The SERVLET_WEBAPPS_DIR, SERVLET_COPY_JARS, SERVLET_SINGLE_DIR_DEPLOY, and SERVLET_SINGLE_DIR_DEPLOY_LICENSE variables perform the same function described for Project Builder's servlet-related build settings earlier.

This is how you set up the SERVLET_WEBAPPS_DIR variable in Project Builder WO:

```
export SERVLET_WEBAPPS_DIR = C:\Tomcat\webapps
```

You can test the servlet by setting the SERVLET_WEBAPPS_DIR build setting to the path of your servlet container's application deployment directory and building the project. Before you build, you can edit Main.wo using WebObjects Builder to add a message to the page, such as Hello. I'm a servlet. When Project Builder finishes building the application, it places the Hello.war file in your servlet container's application deployment directory. The contents of the Hello.war file are listed in Listing 2-1.

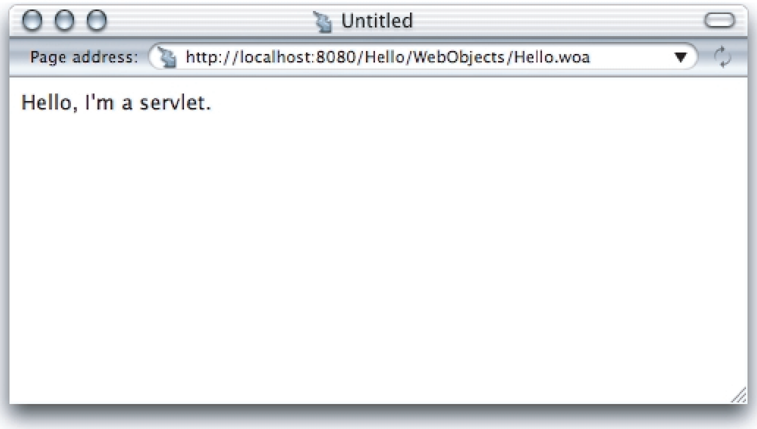
Listing 2-1 Contents of `Hello.war` file

```
Hello/
    META-INF/
        MANIFEST.MF
    WEB-INF/
        classes/
        lib/
            JavaWOJSPServlet_client.jar
        tlds/
            WOtaglib_1_0.tld
        web.xml
```

After restarting your servlet container you can connect to the Hello application through a Web browser. By default, the connection URL is

```
http://host:port/AppName/WebObjects/AppName.woa
```

where `host` is the computer where the servlet container is running and `port` is the port the container runs on. Table 2-1 lists the default host and port for Tomcat, WebLogic, and WebSphere.



http://localhost:8080/Hello/WebObjects/Hello.woa

Hello, I'm a servlet.

Table 2-1 Default host and port in the supported servlet containers

Container	Host	Port
Tomcat	`localhost`	`8080`
WebLogic	`localhost`	`7001`
WebSphere	`localhost`	`9080`

Adding Servlet Support to an Existing Application

To add servlet support to an existing application all you need to do is add the JavaWOJSPServlet framework to your project and re-build it. On Mac OS X, follow these steps:

1. Open the project you want to add servlet support to in Project Builder.
2. Add the JavaWOJSPServlet framework.
    a. Select the Frameworks group from the Groups & Files list.

   b. Choose Project > Add Frameworks.

      A sheet appears with the Frameworks folder selected.

   c. Select JavaWOJSPServlet.framework from the file list, and click Open.

   d. Select Application Server from the target list, and click Add.

      Notice that the Servlet Resources folder is added to the Resources group.

3. Build the project using the Deployment build style.

4. Copy the WAR file or deployment directory in the `build` directory of your project to the application deployment directory of your servlet container.

   You can avoid this step by setting `SERVLET_WEBAPPS_DIR` to the path of your servlet container's application-deployment directory. When using SSDD, you have to add your WebObjects Deployment license number to the project, as explained in "Deploying a Servlet" (page 15).

5. If necessary, restart your servlet container.

The servlet should now be available through your servlet container.

On Windows, follow these steps:

1. Open the project you want to add servlet support to in Project Builder WO.

2. Add the JavaWOJSPServlet framework.

   a. Select the Frameworks bucket.

   b. Choose Project > Add Files.

   c. If necessary, navigate to the `\Apple\Library\Frameworks` directory (the directory should be selected by default).

   d. Select `JavaWOJSPServlet.framework` from the file list and click Open.

   e. Add the servlet-support variables to the `Makefile.preamble` file. One way to do this is by creating a new project with servlet support and copying its servlet-related variables to the `Makefile.preamble` in the project you're modying.

3. Re-build the project.

4. If necessary, copy the WAR file or single deployment directory in the project's `build` directory to the application deployment directory of your servlet container. On Windows, the WAR file or single deployment directory is located at the top level of the project's directory.
5. If necessary, restart your servlet container.

Servlet Single Directory Deployment

As mentioned earlier, Servlet Single Directory Deployment (SSDD) allows you to create an application directory that you can deploy on a computer on which WebObjects is not installed.

To deploy an application using SSDD, you do the following:

1. Set the `SERVLET_SINGLE_DIRECTORY_DEPLOY` build setting to `YES`.
2. Enter your WebObjects Deployment license as the value of the `SERVLET_SINGLE_DIRECTORY_DEPLOY_LICENSE` build setting.

When you build the application, Project Builder creates a directory named after the project. Listing 2-2 lists the contents of the Hello deployment directory.

Listing 2-2 Contents of the `Hello` single deployment directory

```
Hello/
    WEB-INF/
        classes/
        Extensions/                                                //1
        Hello.woa
        lib/
            JavaWOJSPServlet_client.jar
        Library/                                                   //2
            Frameworks/
        LICENSE                                                    //3
        tlds/
            WOtaglib_1_0.tld
        web.xml
```

The following list explains the numbered items in Listing 2-2.

1. The `Extensions` directory contains the JAR files in `/Library/WebObjects/Extensions`.
2. The `Library` directory contains the frameworks in the Frameworks group of the Groups & Files list of the project.
3. The `LICENSE` file contains the WebObjects Deployment license agreement.

Cross-Platform Deployment

To support cross-platform deployment, WebObjects uses three variables that tell the servlet container at runtime where to find WebObjects frameworks (directories with the `.framework` extension) and the WOA bundles (bundles with the extension `.woa`):

- `WOROOT` indicates the path where WebObjects frameworks are installed. On Mac OS X, for example, WebObjects frameworks are located in the `/System/Library/Frameworks` directory and `WOROOT` is set to `/System`. On Windows, `WOROOT` could be set to `C:\Apple`, and on Solaris it may be `/opt/Apple`.
- `LOCALROOT` indicates the path where local frameworks are installed. On Mac OS X, these frameworks are located in the `/Library/Frameworks` directory, and `LOCALROOT` is set to `/`. On Windows, `LOCALROOT` may be set to `C:\Apple\Local`, while on Solaris it could be `/opt/Apple/Local`.
- `WOAINSTALLROOT` specifies the location of WOA bundles. On Mac OS X, the default is `/Library/WebObjects/Applications`.

When you deploy the WAR file of your servlet on a computer where the framework and WOA files are in different locations from the default ones, you can specify the correct paths using the variables described above. You can accomplish this in two ways:

- configuring the application's deployment descriptor

- configuring the servlet container

Note: Single directory deployments, described in "Servlet Single Directory Deployment" (page 21), are platform independent.

Configuring the Deployment Descriptor

The deployment descriptor of a servlet is the `web.xml` file, located in the `WEB-INF` directory of the WAR file. This file is generated from the `web.xml.template` file in your project.

To configure your application's deployment descriptor during development, you edit the `web.xml.template` file. Alternatively, you can edit the `web.xml` file of the WAR file (after expanding the WAR file). Locate the `<param-name>` tags for the appropriate variables, and set the value for their corresponding `<param-value>` tag.

This is an example of a `web.xml.template` file on Windows:

```
<web-app>
    <context-param>
        <param-name>WOROOT</param-name>
        <param-value>C:\WebObjectsFrameworks</param-value>
    </context-param>
    <context-param>
        <param-name>LOCALROOT</param-name>
        <param-value>C:\Apple\Local</param-value>
    </context-param>
    <context-param>
        <param-name>WOAINSTALLROOT</param-name>
        <param-value>C:\WebObjectsApplications</param-value>
    </context-param>
    ...
</web-app>
```

You expand the WAR file by executing the following commands in your shell editor:

```
mkdir filename
jar -xvf filename.war
```

When you're done editing the `web.xml` file, you re-create the WAR file by executing

```
jar -cvf fileName.war .
```

Configuring the Servlet Container

This method allows your settings to be propagated to all applications and it overrides the values set in the deployment descriptor. Using this approach, you can deploy WebObjects applications without worrying about each application's configuration. You can configure the servlet container in two ways:

- editing the launch script of the servlet container
- defining environment variables

This is an example the launch script in Tomcat (`startup.sh`):

```
#! /bin/sh
...
$JAVACMD $TOMCAT_OPTS -DWOROOT=/System -DLOCALROOT=/
-DWOAINSTALLROOT=/Library/WebObjects/Applications
-Dtomcat.home=${TOMCAT_HOME}  org.apache.tomcat.startup.Tomcat "$@" &

BASEDIR='dirname $0'
$BASEDIR/tomcat.sh start "$@"
```

This is an example of the launch-script format in WebLogic (`startWLS.sh`):

```
"${JAVA_HOME}/bin/java" ${JAVA_VM} ${MEM_ARGS}
-classpath ${CLASSPATH}"
-Dweblogic.Name=myserver
-Dbea.home="/opt/bea"
"-DWOROOT=/opt/Apple"
"-DLOCALROOT=/opt/Apple/Local"
"-DWOAINSTALLROOT=/applications/production"
-Dweblogic.management.username=${WLS_USER}
-Dweblogic.management.password=${WLS_PW}
-Dweblogic.ProductionModeEnabled=${STARTMODE}
-Djava.security.policy="${WL_HOME}/server/lib/weblogic.policy"
weblogic.Server
```

This is how you would define environment variables using the bash or zsh shell editors:

```
% export TOMCAT_OPTS="-DWOROOT=/System -DWOAINSTALLROOT=/WebObjects/
Applications -DLOCALROOT=/"
```

And this is how you would do it using the csh shell editor:

```
% setenv TOMCAT_OPTS "-DWOROOT=/System -DWOAINSTALLROOT=/WebObjects/
Applications -DLOCALROOT=/"
```

Installing Servlets in WebSphere

To install a single deployment directory you need to create a WAR file from the directory. Execute the following commands to create the WAR file:

```
cd <path-to-project>/build/AppName
jar cvf AppName.war .
```

To install a WAR file, perform these steps using `console`:

1. Choose Nodes > Server > Enterprise Apps > Install.
2. Navigate to the WAR file's location.
3. Enter the application's name in the App Name text input field; for example, `MyApp`.
4. Enter the context name for the application in the Context Root text input field; for example, `/MyApp`.

C H A P T E R 3

JavaServer Pages

JavaServer Pages (JSP) is a specification that describes what a servlet-based content creation system should do. One of its main purposes is to facilitate the creation of dynamic Web pages.

You can directly access WebObjects components in your JSP pages. These components can be WOComponents or WODirectActions. This allows you to create JSP-based applications that take advantage of WebObjects technologies, such as Enterprise Objects.

When your servlet container receives a request addressed to a JSP page, the container reads the `.jsp` file and compiles it into a workhorse servlet that processes the HTTP requests and produces responses to them.

This chapter addresses the following topics:

- "JavaServer Page Writing Guidelines" (page 28) introduces the custom tag library that your JSP pages must include to be able to access WebObjects components.
- "Developing a JavaServer Page–Based Application" (page 30) walks you through the steps needed to create a simple JSP-based application.
- "Passing Data From a JSP Page to a Component" (page 33) explains what you need to do in order to pass data from a JSP page to a WebObjects component or direct action.
- "Using WebObjects Classes in a JSP Page" (page 37) shows you how to write JSP pages that use WebObjects classes.
- "Using Direct Actions in JSP Pages" (page 40) explains how to use a direct action in a JSP page.
- "Custom-Tag Reference" (page 45) provides a detailed explanation for each of the tags defined in the custom tag library.

JavaServer Page Writing Guidelines

To be able to use WebObjects components in your JSP pages, you have to include the `WOtaglib_1_0.tld` custom tag library. It's located in `/System/Library/Frameworks/JavaWOJSPServlet.framework/Resources`.
This custom tag library uses the tag library descriptor format defined in a DTD (Document Type Definition) from Sun. This DTD is available at http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd.

The elements you use in your JSP pages have the form `<wo:`*elementName*`>`. *elementName* indicates the type of element you want to use. For example, to use a `component` element within a JSP page, you add code like the following to the `.jsp` file:

```
<wo:component ...>
    ...
</wo:component>
```

Version 1.0 of the custom tag library defines five tags as described in Table 3-1.

Table 3-1 Custom elements defined in WOtaglib_1_0.tld

Element	Children	Description
`wo:component`	`binding` `extraHeader`	Top-level element. Specifies the component that is used in the JSP page.
`wo:directAction`	`formValue` `extraHeader`	Top-level element. Specifies the direct action that is used in the JSP page.
`wo:extraHeader`	None	Specifies the extra HTTP headers to be passed to the component or direct action.
`wo:binding`	None	Specifies the key-value pair to be passed to the containing `wo:component` for binding.
`wo:formValue`	None	Specifies the form value to be passed to the containing `wo:directAction`.

For detailed information on the WebObjects custom tag library, see "Custom-Tag Reference" (page 45).

To use the `wo:component` or `wo:directAction` elements on a JSP page, you must add the following directive to the page:

```
<%@ taglib uri="/WOtaglib_1_0.tld" prefix="wo" %>
```

When you need to access WebObjects classes or objects from your JSP page, you need to copy all the framework and application JAR files necessary into the WAR file or single deployment directory. You accomplish this by calling the `initStatics` method of the WOServletAdaptor class:

```
<% WOServletAdaptor.initStatics(application); %>
```

Note that you need to invoke the `initStatics` method only once during the lifetime of an application. Furthermore, the method is invoked automatically anytime `wo:component` or `wo:directAction` elements are used in a JSP page.

You also need to import the appropriate packages before using the classes with the `import` attribute of the page directive in your JSP page:

```
<%@ page import = "com.webobjects.jspservlet.*" %>
```

These directives need to be performed only once per page. However, additional invocations have no ill effect. Referencing classes directly is useful when using components that require binding values. For example, a WORepetition whose `list` attribute is bound to an array of enterprise-object instances.

This is an example of a `directAction` definition:

```
<wo:directAction actionName="random" className="DirectAction">
    <wo:formValue key = "formKey" value = '<%= "formValue" %>'/>
    <wo:extraHeader key = "headerKey" value = '<%= "headerValue" %>'/>
</wo:directAction>
```

This is an example of a `component` definition:

```
<wo:component className="MyImageComponent">
    <wo:binding key="filename" value='<%= "start.gif" %>' />
</wo:component>
```

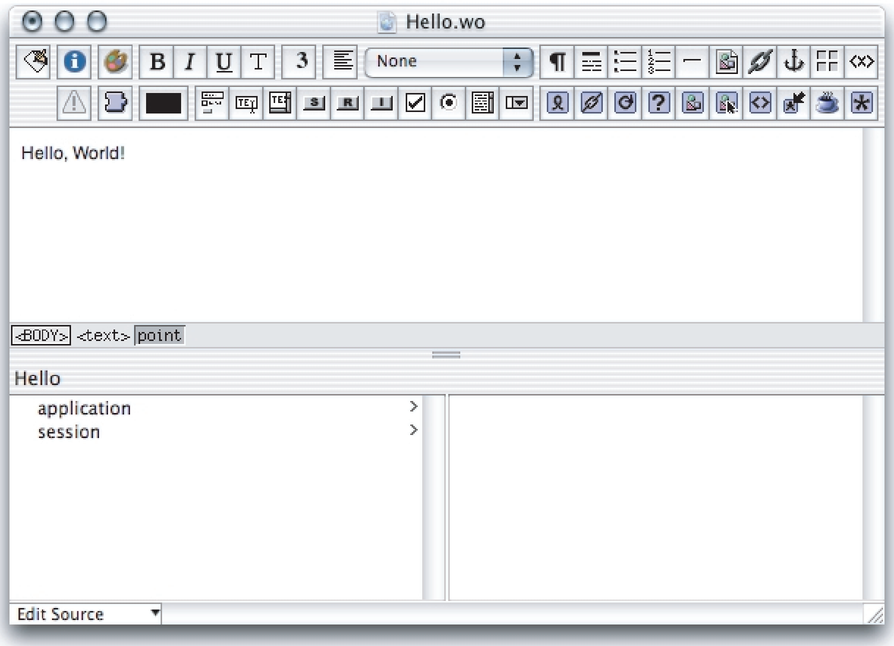
To embed dynamic elements in a JSP page, such WOConditional and WORepetition, you have to wrap them in a WebObjects component, which you then use in your JSP page.

Developing a JavaServer Page–Based Application

This section shows you how to create a simple JSP-based WebObjects application. In it you learn how to use `wo:component` elements in a JSP page.

1. Launch Project Builder and create a WebObjects Application project called `JSP_Example`.
2. In the J2EE Integration pane of the Project Builder Assistant, select "Deploy in a servlet container."
3. In Project Builder, create a component called `Hello` (make sure you assign it to the Application Server target). Edit the component using WebObjects Builder so that it looks like Figure 3-1.

Figure 3-1 JSP_Example project—the Hello component



Hello, World!

4. Set the servlet application directory. (See "Deploying a Servlet" (page 15) for details.)
5. In the Finder, navigate to the Servlet Resources folder, located in the JSP_Example folder, and create a folder called jsp.
6. Using a text editor, create a file with the following contents:

```
<%-- Welcome.jsp --%>

<%@ taglib uri="/WOtaglib" prefix="wo" %>

<HTML>

<HEAD>
    <TITLE>Welcome to JavaServer Pages in WebObjects</TITLE>
</HEAD>
```

```
<BODY>
    <wo:component className="Hello">
    </wo:component>
</BODY>

</HTML>
```

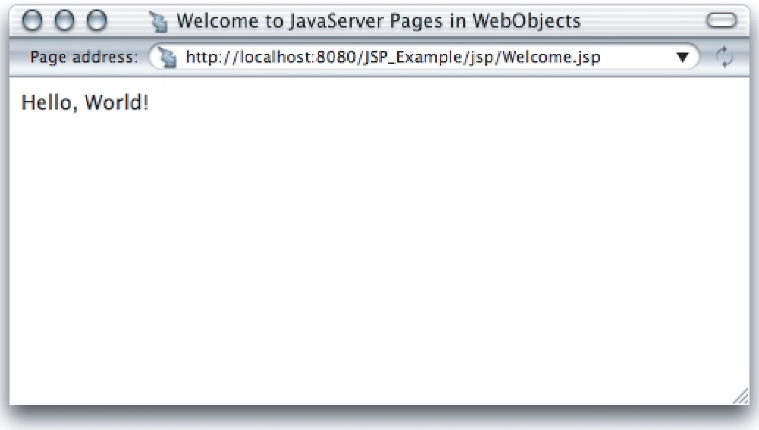
7. Save the file as `Welcome.jsp` in the `jsp` directory.
8. Build the JSP_Example project (if necessary, restart your servlet container).

You should now be able to connect to your application. In Tomcat, you use the following URL:

```
http://localhost:8080/JSP_Example/jsp/Welcome.jsp
```

A page similar to the one in Figure 3-2 should appear in your browser. (Otherwise, consult your servlet container's documentation to make sure that it's configured properly.)

Figure 3-2 JSP_Example project—the output of Welcome.jsp


Welcome to JavaServer Pages in WebObjects
Page address:
http://localhost:8080/JSP_Example/jsp/Welcome.jsp

Hello, World!

Passing Data From a JSP Page to a Component

In this section, you expand the JSP_Example project to include

- a new component called FavoriteFood
- a JSP page, called DiningWell, that uses the Hello and FavoriteFood components to generate its output

The FavoriteFood component contains two attributes: `visitorName` and `favoriteFood`. When the DiningWell workhorse servlet receives a request, it passes two strings to the FavoriteFood component. The FavoriteFood component then uses those strings to render its HTML code.

1. Using a text editor, create a file with the following contents:

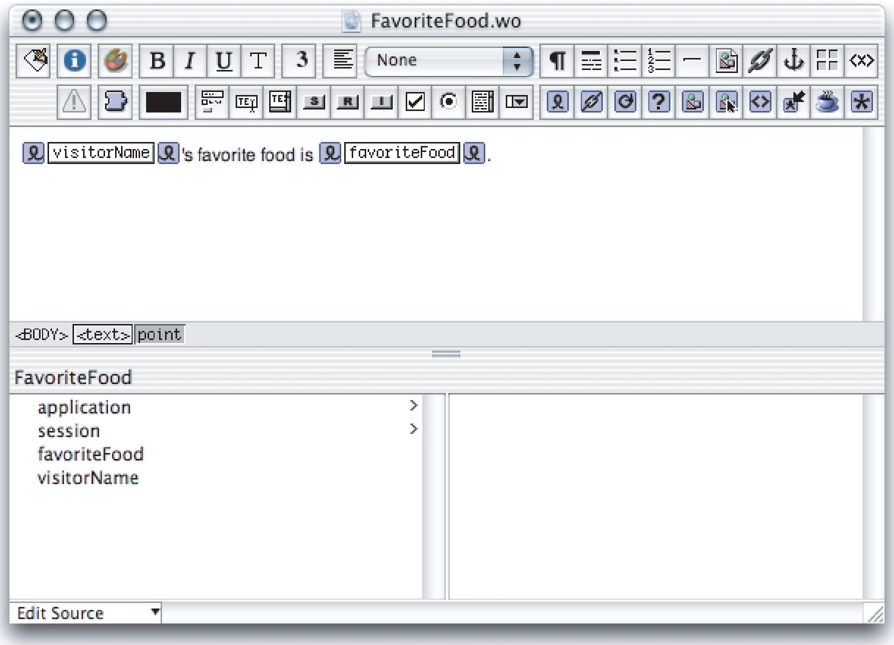
```
<%-- DiningWell.jsp --%>

<%@ taglib uri="/WOtaglib" prefix="wo" %>

<HTML>

<HEAD>
    <TITLE>What to eat?</TITLE>
</HEAD>

<BODY>
    <wo:component className="Hello" />
    <P><P>
    <wo:component className="FavoriteFood" bodyContentOnly="true">
        <wo:binding key="visitorName" value='<%= "Worf" %>' />
        <wo:binding key="favoriteFood" value='<%= "gagh" %>' />
    </wo:component>
</BODY>

</HTML>
```

Note that in this case the `bodyContentOnly` attribute of the `wo:component` element is set to `true` (this is the default, so you don't need to specify a value for it). This allows you to define the FavoriteFood component as "Full document" (the default setting in WebObjects Builder) instead of "Partial document." This way, the component can be viewed as a Web page on its own and as a component within a JSP page.

For faster processing, you can set the `bodyContentOnly` attribute to `false` if you are certain that the component only includes the `BODY` element and not the `HTML` element.

2. Save the file as `DiningWell.jsp` in `JSP_Example/Servlet Resources/jsp`.
3. In Project Builder, create a component called `FavoriteFood` (make sure you assign it to the Application Server target).
4. Edit the component using WebObjects Builder so that it looks like Figure 3-3. Make sure to add accessor methods to the `visitorName` and `favoriteFood` String keys. Also, ensure that the FavoriteFood component is set to "Full document".

Figure 3-3 JSP_Example project—the DiningWell component



When you're done `FavoriteFood.java` should look like Listing 3-1.

Listing 3-1 FavoriteFood.java

```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoaccess.*;

public class FavoriteFood extends WOComponent {
    protected String visitorName;
    protected String favoriteFood;

    public FavoriteFood(WOContext context) {
        super(context);
    }

    public String visitorName() {
        return visitorName;
    }
    public void setVisitorName(String newVisitorName) {
        visitorName = newVisitorName;
    }

    public String favoriteFood() {
        return favoriteFood;
    }
    public void setFavoriteFood(String newFavoriteFood) {
        favoriteFood = newFavoriteFood;
    }
}
```

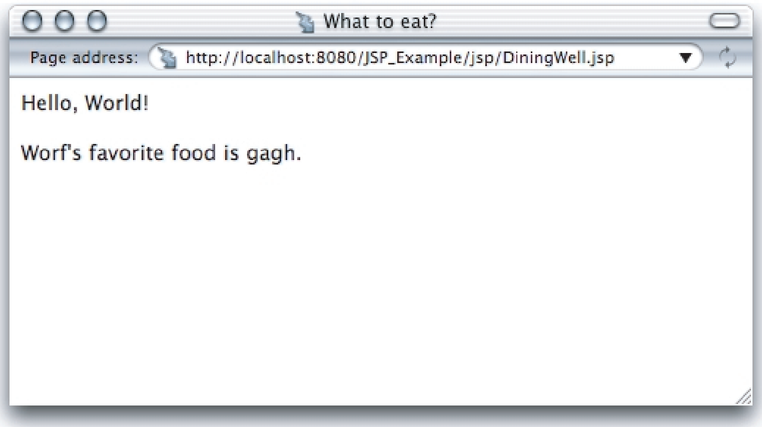
5. Build the project and restart your servlet container, if necessary.

If you're using Tomcat, you can view the new page in your browser with this URL

```
http://localhost:8080/JSP_Example/jsp/DiningWell.jsp
```

The Web page should look like Figure 3-4.

Figure 3-4 JSP_Example project—the output of DiningWell.jsp


What to eat?
Page address: http://localhost:8080/JSP_Example/jsp/DiningWell.jsp

```
Hello, World!

Worf's favorite food is gagh.
```

This is the HTML code your Web browser receives (the listing is indented for easy reading):

```
<HTML>
    <HEAD>
        <TITLE>What to eat?</TITLE>
    </HEAD>

    <BODY>
        Hello, World!
        <P><P>
        Worf's favorite food is gagh.
    </BODY>
</HTML>
```

Using WebObjects Classes in a JSP Page

This section continues work on the JSP_Example project. It explains how to write a JSP page that makes use of two WebObjects classes, NSArray and NSMutableArray, to pass information to a component called MusicGenres.

1. Using a text editor, create a file with the contents of Listing 3-2.

Listing 3-2 InternetRadio.jsp file

```
<%-- InternetRadio.jsp --%>

<%@ taglib uri="/WOtaglib" prefix="wo" %>

<%-- Import statements --%>
<%@ page import="com.webobjects.foundation.*" %>
<%@ page import="com.webobjects.jspservlet.*" %>

<%-- Initialize WebObjects-to-servlet-container integration system --%>
<%
    WOServletAdaptor.initStatics(application);
```

```
%>

<%-- Create musical-genre list --%>
<%
    NSMutableArray genres = new NSMutableArray();
    genres.addObject(new String("Classical"));
    genres.addObject(new String("Country"));
    genres.addObject(new String("Eclectic"));
    genres.addObject(new String("Electronica"));
    genres.addObject(new String("Hard Rock/Metal"));
    genres.addObject(new String("Hip-Hop/Rap"));
    genres.addObject(new String("Jazz"));
%>

<HTML>

<HEAD>
    <TITLE>Music Available on Internet Radio Stations</TITLE>
</HEAD>

<BODY>
    <wo:component className="MusicGenres" bodyContentOnly="true">
        <wo:binding key="genres" value='<%= genres %>' />
    </wo:component>
</BODY>

</HTML>
```

Note the invocation of the `initStatics` method of the WOServletAdaptor class. It performs the initialization of objects needed to integrate WebObjects with your servlet container (for example, adding a WOSession object to the JSPSession object).

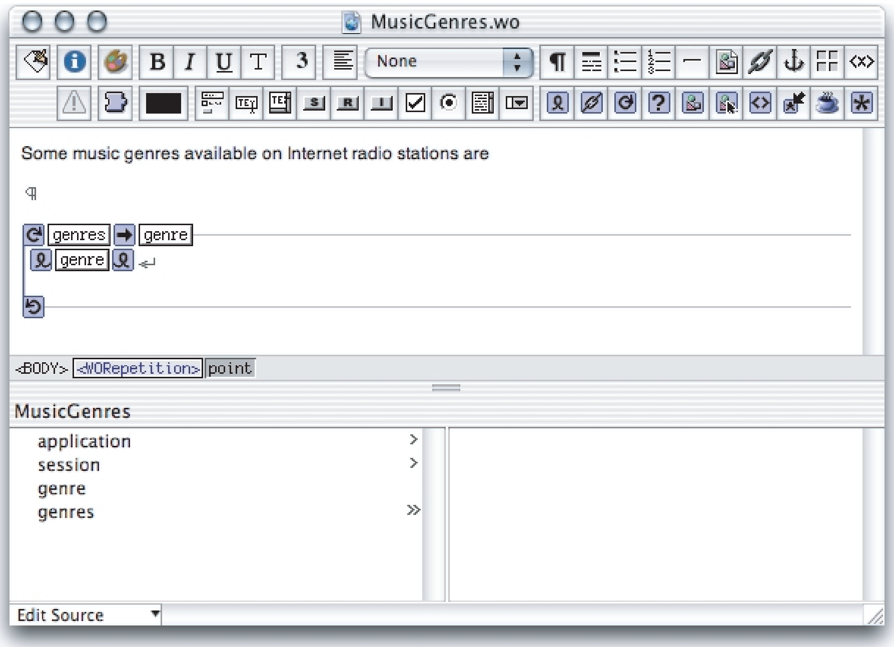
2. Save the file as `InternetRadio.jsp` in the `JSP_Example/Servlet Resources/jsp` directory.
3. In Project Builder, create a component called `MusicGenres` (make sure you assign it to the Application Server target).
4. Add the `genres` and `genre` keys to MusicGenres using WebObjects Builder. `genres` is an array of Strings and `genre` is a String. Add a setter method for `genres`.

   Alternatively, you can add the following code to `MusicGenres.java`:

```
protected String genre;

/** @TypeInfo java.lang.String */
protected NSArray genres;

public void setGenres(NSArray newGenres) {
    genres = newGenres;
}
```

5. Edit the component using WebObjects Builder so that it looks like Figure 3-5.

Figure 3-5 JSP_Example project—the MusicGenres component



Some music genres available on Internet radio stations are

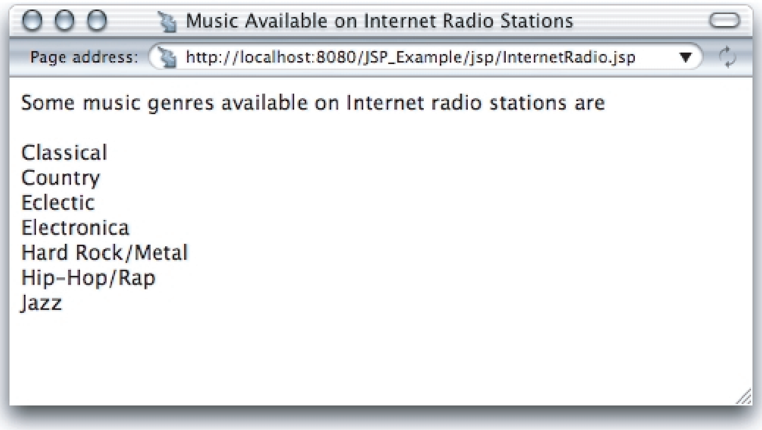
6. Tell Project Builder to copy the necessary WebObjects classes to the WAR file or single deployment directory by setting the `SERVLET_COPY_JARS` build setting to `YES`.
7. Build the application and restart your servlet container, if necessary.

To view the output of the InternetRadio JSP page in Tomcat use the following URL:

```
http://localhost:8080/JSP_Example/jsp/InternetRadio.jsp
```

You should see a page like the one in Figure 3-6.

Figure 3-6 JSP_Example project—the output of InternetRadio.jsp


Music Available on Internet Radio Stations
Page address:
http://localhost:8080/JSP_Example/jsp/InternetRadio.jsp

Some music genres available on Internet radio stations are

Classical
Country
Eclectic
Electronica
Hard Rock/Metal
Hip-Hop/Rap
Jazz

Using Direct Actions in JSP Pages

This section shows you how to create a WebObjects component called `FoodInquiry` that contains a WOForm element with two WOTextFields and a WOSubmitButton. The FoodInquiry page is displayed by a direct action, which itself is invoked by a JSP page that provides the FoodInquiry component with initial values for its form elements using `wo:formValue` elements.

1. Using a text editor, create a file with the following contents:

```
<%-- LogIn.jsp --%>

<%@ taglib uri="/WOtaglib" prefix="wo" %>
```

```
<wo:directAction actionName="login" className="DirectAction"
bodyContentOnly="false">
    <wo:formValue key="VisitorName" value='<%= "enter name" %>' />
    <wo:formValue key="FavoriteFood" value='<%= "enter food" %>' />
</wo:directAction>
```

2. Save the file as `LogIn.jsp` in `JSP_Example/Servlet Resources/jsp`.
3. In Project Builder, create a component called FoodInquiry (make sure you assign it to the Application Server target).
4. Add the `visitorName` and `favoriteFood` String keys to the component (create accessor methods). Also add the `showFavoriteFood` action returning the FavoriteFood component.

   When you're done, `FoodInquiry.java` should look like Listing 3-3. (Note that if you use WebObjects Builder to add the keys and the action, you need to add a couple of lines of code to the `showFavoriteFood` method.

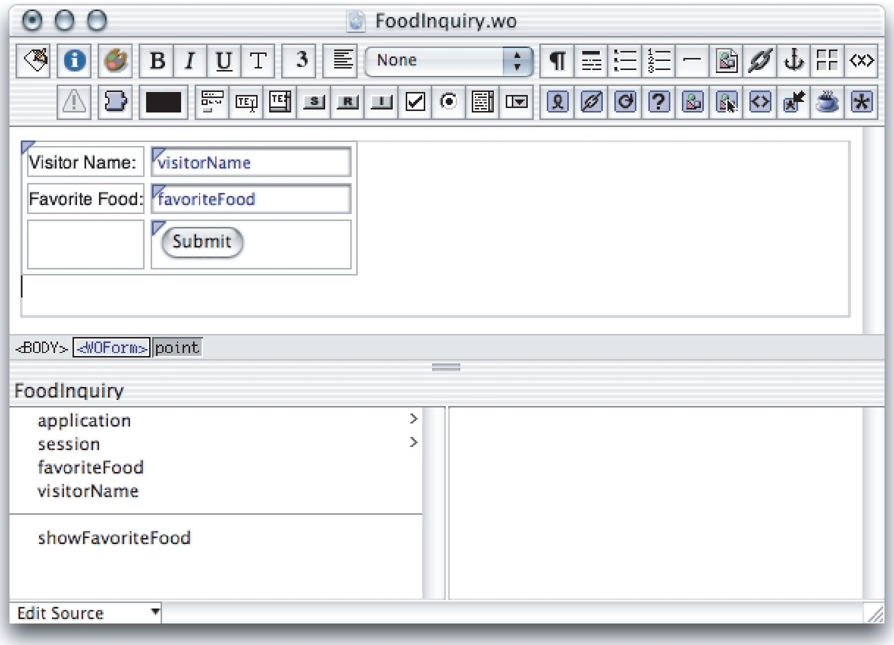
Listing 3-3 FoodInquiry.java

```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoaccess.*;

public class FoodInquiry extends WOComponent {
    protected String visitorName;
    protected String favoriteFood;

    public FoodInquiry(WOContext context) {
        super(context);
    }

    public FavoriteFood showFavoriteFood() {
        FavoriteFood nextPage =
(FavoriteFood)pageWithName("FavoriteFood");

        // Set the properties of the FavoriteFood component.
        nextPage.setVisitorName(visitorName);
        nextPage.setFavoriteFood(favoriteFood);
```

```
            return nextPage;
        }

        public String visitorName() {
            return visitorName;
        }
        public void setVisitorName(String newVisitorName) {
            visitorName = newVisitorName;
        }

        public String favoriteFood() {
            return favoriteFood;
        }
        public void setFavoriteFood(String newFavoriteFood) {
            favoriteFood = newFavoriteFood;
        }
    }
```

5. Edit the component using WebObjects Builder so that it looks like Figure 3-7.

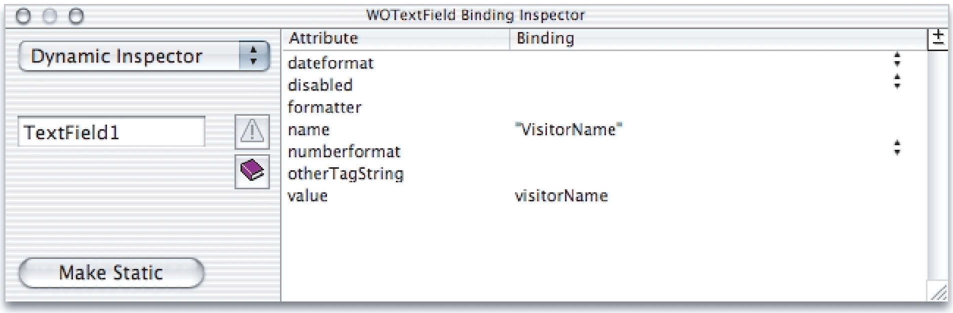
Figure 3-7 JSP_Example project—the FoodInquiry component



a. Bind the Submit button to the `showFavoriteFood` action.

b. Enter `Food Inquiry` as the component's title.

c. Enter "`VisitorName`" as the value for the `name` attribute of the WOTextField that corresponds to the Visitor Name label.



Attribute	Binding
dateformat	
disabled	
formatter	
name	"VisitorName"
numberformat	
otherTagString	
value	visitorName

d. Enter "`FavoriteFood`" as the value for the `name` attribute of the WOTextField that corresponds to the Favorite Food label.

6. Add the `loginAction` method (listed below) to the DirectAction class.

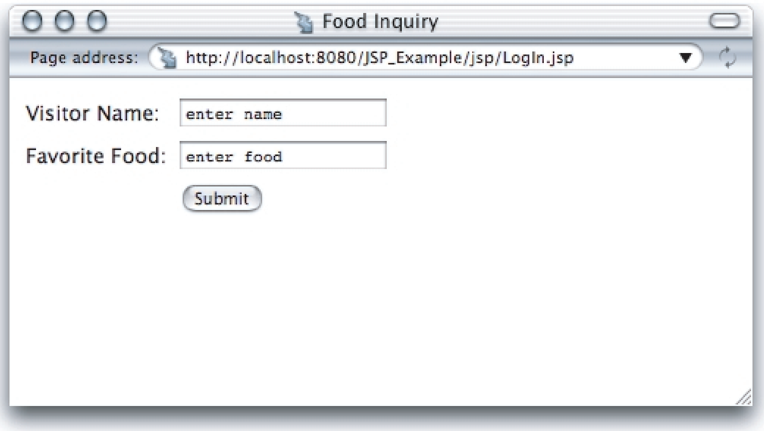
```
public WOActionResults loginAction() {
    FoodInquiry result = (FoodInquiry)pageWithName("FoodInquiry");

    // Get form values.
    String visitorName = request().stringFormValueForKey("VisitorName");
    String favoriteFood= request().stringFormValueForKey("FavoriteFood");

    // Set the component's instance variables.
    result.setVisitorName(visitorName);
    result.setFavoriteFood(favoriteFood);

    return result;
}
```

To view the output of the LogIn JSP page, use the following URL (restart your servlet container, if necessary):

```
http://localhost:8080/JSP_Example/jsp/LogIn.jsp
```

You should see a page like the one in Figure 3-8.

Figure 3-8 JSP_Example project—the output of LogIn.jsp


Food Inquiry
Page address: http://localhost:8080/JSP_Example/jsp/LogIn.jsp
Visitor Name: enter name
Favorite Food: enter food
Submit

Custom-Tag Reference

The following sections provide details about the custom WebObjects JSP tags that `WOtaglib_1_0.tld` defines.

wo:component

You use this element to embed a WebObjects component within a JSP page. Table 3-2 describes its attributes.

Table 3-2 Attributes of the `wo:component` element

Attribute	Required	Description
`className`	Yes	Class name of the WebObjects component.
`bodyContentOnly`	No	Indicates whether the JSP page requires only the body content of the response (without `<HTML>` and `</HTML>` tags). Values: `true` or `false`. Default: `true`.
`mergeResponseHeaders`	No	Indicates whether the WOResponse headers are to be merged with the ServletResponse headers. Values: `true` or `false`. Default: `false`.

wo:directAction

You use this element to embed a direct action within a JSP page. Table 3-3 describes its attributes.

Table 3-3 Attributes of the `wo:directAction` element

Attribute	Required	Description
`actionName`	Yes	Specifies the direct-action name.
`className`	No	Specifies the direct-action class name. Default: `DirectAction`.

Table 3-3 Attributes of the `wo:directAction` element

Attribute	Required	Description
`contentStream`	No	Specifies the source of the request's content; it must be an InputStream (or a subclass).
`bodyContentOnly`	No	Indicates whether the JSP page requires only the body content of the response (without `<HTML>` and `</HTML>` tags). Values: `true` or `false`. Default: `true`.
`mergeResponseHeaders`	No	Indicates whether the WOResponse headers are to be merged with the ServletResponse headers. Values: `true` or `false`. Default: `false`.

wo:extraHeader

The `wo:extraHeader` element specifies a key-value pair to be passed to the component or direct action as an HTTP header. A `wo:extraHeader` element has to be used for each header value; you can pass multiple values for one header by using the same value for the `key` attribute in multiple `wo:extraHeader` elements. If the value is not `null`, it must be a String. Otherwise, the corresponding header is removed from the request before it's passed to the component or direct action. Table 3-4 describes the attributes of this element.

Table 3-4 Attributes of the `wo:extraHeader` element

Attribute	Required	Description
`key`	Yes	Specifies the HTTP header.
`value`	Yes	Specifies the value for the HTTP header.

wo:binding

This element specifies a key-value pair to be passed to the component to satisfy one of its bindings. You need a `wo:binding` element for each of the component's bindings. Table 3-5 describes its attributes.

Table 3-5 Attributes of the `binding` element

Attribute	Required	Description
`key`	Yes	Specifies the component's binding.
`value`	Yes	Specifies the value for the binding.

wo:formValue

This element specifies a key-value pair to be passed to the direct action in a query string; it must be a String. You need a `wo:formValue` for each item in the form. Table 3-6 describes the attributes of this element.

Table 3-6 Attributes of the `formValue` element

Attribute	Required	Description
`key`	Yes	Specifies the form element.
`value`	Yes	Specifies the value for the form element.

Special Issues

There are two special issues regarding JSP and Servlet support in WebObjects that you should keep in mind: deploying more than one WebObjects application within a single container and updating existing servlet-based WebObjects applications to future versions of WebObjects. The sections below explain how to address both of these.

Deploying Multiple WebObjects Applications in a Single Servlet Container

Having more than one WebObjects application file in a servlet container is relatively safe. However, as each application launches, it pushes the values of its launch properties to the system properties (the properties maintained by the `java.lang.System` class. Therefore, the WebObjects application launched last within a servlet container overrides the properties set by previously launched WebObjects applications in that container.

The solution is to ensure WebObjects applications deployed within one servlet container use the same values for the following properties:

- `NSProjectSearchPath`
- `WOAdaptorURL`
- `WOAdditionalAdaptors`
- `WOAllowsCacheControlHeader`
- `WOAllowsConcurrentRequestHandling`

- `WOApplicationBaseURL`
- `WOAutoOpenClientApplication`
- `WOAutoOpenInBrowser`
- `WOCachingEnabled`
- `WOContextClassName`
- `WODebuggingEnabled`
- `WOFrameworksBaseURL`
- `WOIncludeCommentsInResponse`
- `WOMaxHeaders`
- `WOMaxIOBufferSize`
- `WOSMTPHost`
- `WOSessionStoreClassName`

Updating Servlet-Based Applications to Future Versions of WebObjects

If future versions of WebObjects include changes to the JSP and Servlet system, it is likely that you need to update the `web.xml.template` file (on Mac OS X) or the `Makefile.preamble` file (on Windows) for existing applications.

To update the `web.xml.template` in a project developed on Mac OS X follow these steps:

1. Open the project you want to update in Project Builder.
2. Create a new WebObjects Application project that includes JSP and Servlet support by choosing "Deploy in a JSP/Servlet Container" in the Enable J2EE Integration pane of the Project Builder Assistant.

3. Copy the contents of the new project's `web.xml.template` file to the `web.xml.template` file of the project you want to update.

   On Mac OS X, if you have made changes to the `web.xml.template` file, you can use FileMerge to keep your modifications in the updated version.

To update a WebObjects application developed on Windows perform the following steps:

1. Open the project you want to update in Project Builder WO.
2. Create a new Java WebObjects Application project that includes JSP and Servlet support by choosing "Deploy in a JSP/Servlet Container" in the Enable J2EE Integration pane of the WebObjects Application Wizard.
3. Copy the contents of the new project's `Makefile.preamble` file to the `Makefile.preamble` file of the project you want to update.

In addition, you should also rebuild your projects (regenerate the WAR files or single deployment directories) to update the applications with the latest version of the WebObjects frameworks.

Document Revision History

Table B-1 describes the revisions to *Inside WebObjects: JavaServer Pages and Servlets*.

Table B-1 Document revision history

Date	Notes
September 2002	Project examples now in `/Developer/Documentation/WebObjects/JSP_and_Servlets/projects`.
	Added information on Servle Single Directory Deployment.
	Revised for WebObjects 5.2.
	Document name changed to *Inside WebObjects: JavaServer Pages and Servlets*.
January 2002	Document published as *Inside WebObjects: Developing Applications Using JavaServer Pages and Servlets*.

A P P E N D I X   B

Document Revision History

Glossary

bundle On Mac OS X systems, a bundle is a directory in the file system that stores executable code and the software resources related to that code. The bundle directory, in essence, groups a set of resources in a discrete package.

CGI (Common Gateway Interface) A standard for communication between external applications and information servers, such as HTTP or Web servers.

component An object (of the WOComponent class) that represents a Web page or a reusable portion of one.

data-source adaptor A mechanism that connects your application to a particular database server. For each type of server you use, you need a separate adaptor. WebObjects provides an adaptor for databases conforming to JDBC.

deployment descriptor XML file that describes the configuration of a Web application. It's located in the `WEB-INF` directory of the application's WAR file and named `web.xml`.

HTTP adaptor A process (or a part of one) that connects WebObjects applications to a Web server.

HTTP server, Web server An application that serves Web pages to Web browsers using the HTTP protocol. In WebObjects, the Web server lies between the browser and a WebObjects application. When the Web server receives a request from a browser, it passes the request to the WebObjects adaptor, which generates a response and returns it to the Web server. The Web server then sends the response to the browser.

J2EE (Java 2 Platform, Enterprise Edition) Specification that define a platform for the development and deployment of Web applications. It defines an environment under which enterprise beans, servlets, and JSP pages can share resources and work together.

JAR (Java archive) A file created using the `jar` utility (and saved with the `.jar` extension) that contains all the files that make up a Java application.

JSP (JavaServer Pages) Technology that facilitates the development of dynamic Web pages and Web applications that use existing components, such as JavaBeans and WebObjects components.

Monitor WebObjects application used to configure and maintain deployed WebObjects applications capable of handling multiple applications, application instances, and applications hosts at the same time.

Project Builder Application used to manage the development of a WebObjects application or framework.

request A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the user's Web browser to a Web server that asks for a resource like a Web page.

response A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the Web server to the user's Web browser that contains the resource specified by the corresponding request. The response is typically a Web page.

servlet A Java program that runs as part of a network service, typically an HTTP server and responds to requests from clients. Servlets extend an HTTP server by generating content dynamically.

servlet container Java application that provides a working environment for servlets. It manages the servlet's interaction with its client and provides the servlet access to various Java-based services. Containers can be implemented as standalone HTTP servers, server plug-ins, and components that can be embedded in an application.

TLD (tag library descriptor) XML document that describes a tag library. A JSP container uses the information contained in the TLD file to validate a JSP page's tags.

WAR (Web application archive) A file created using the `jar` utility (and saved with the `.war` extension) that contains all the files that make up a Web application.

WOA (WebObjects application bundle) A bundle that stores all the files needed by a WebObjects application.

wotaskd (WebObjects task daemon) WebObjects Deployment tool that manages the instances on an application host. It's used by Monitor to propagate site configuration changes throughout the site's application hosts.

Web application, Web app File structure that contains servlets, JSP pages, HTML documents and other resources. This structure can be deployed on any servlet-enabled HTTP server.

Index